# Performance Coupling: A Methodology for Predicting Application Performance using Kernel Performance

Jonathan Geisler and Valerie Taylor
Electrical and Computer Engineering
Northwestern University
2145 Sheridan Rd.
Evanston, IL 60208
{geisler,taylor}@ece.nwu.edu

**Abstract**

Traditional performance optimization techniques have focused on finding the kernel in a program that is the most time consuming and attempting to optimize it. We introduce a methodology for measuring and representing the interaction, or *coupling*, between kernels that improves upon the accuracy of the traditional method. Then we demonstrate the benefits of using the new methodology by developing a new hybrid algorithm for the conjugate gradient application that results in fewer cache misses than the other algorithms studied.

## 1 Introduction

Traditional performance optimization techniques have focused on finding the kernel in a program that is the most time consuming and attempting to optimize it. An example of such an optimization entails restructuring an algorithm to increase data reuse (i.e., blocking [6]), thereby reducing cache misses. It is well known that the performance increase that is achieved when optimizing a given kernel in isolation generally does not reflect the performance increase that occurs when the new kernel is included in the larger application [11]. This disparity in performance increase between kernel and full application is due in part to a lack of understanding of how the interaction, or *coupling*, of kernels affects the performance of the application.

In this paper, we present a methodology for measuring this coupling and describe how the measurements can be used to obtain an efficient application. In particular, we measure how kernels within an application perform in isolation and how different kernels perform with the other kernels in the application. The measurements lead to a parameter that is a ratio between the kernels performance together and the separate performances of the kernels summed. The parameter allows the algorithm designer to know how much a kernel interacts with the other kernels in the application. With this ratio and a full understanding of the algorithm, the designer can improve the application performance by changing the kernel so that it can benefit from the work done by previous kernels and/or doing work from which future kernels can benefit.

This work focuses on single processor performance. The methodology generalizes to a parallel processing framework by adding another level of interaction to represent the interconnection network. Future work will examine parallel applications more fully, but we begin with serial applications because of the importance of obtaining optimized serial

codes. Within the single processor, we are looking at interaction that occurs within the memory hierarchy; in particular the first level cache. With the increasing disparity between CPU and memory performance, it is becoming increasingly important to optimize memory hierarchy usage.

In Section 3, we describe the new methodology and how it is used. In Section 4, we illustrate the methodology using a simple example followed by results obtained with the conjugate gradient benchmark in Section 5. The original coupling results are used to illustrate the development of a new data structure that reduces the total number of level one cache misses slightly.

## 2 Related Work

In [7] and [8], Rafael Saavedra did much work characterizing various benchmarks by decomposing them to high level Fortran-like statements. He then counted the number of times each statement occurred in the program. By measuring the execution time of each statement on various target machines, he was able to predict the total execution time of the benchmarks by multiplying the statement execution times by the number of times it occurred and then summing that product over all statements. Our work complements Rafael's work in quantifying and understanding the interaction between kernels.

Significant work has been done to develop models for parallel execution. Examples of these are LogP [3], BSP [13], and CRAM [10]. While the models focus on the system effects, with a major emphasis on interprocessor communication, the memory hierarchy is not represented. Only the PMH model [1] parameterizes the processor memory hierarchy. Interaction between kernels is not represented or considered, which is the focus of this paper.

## 3 Proposed Methodology

### 3.1 Description

Our methodology focuses on extracting the performance coupling between pairs of kernels ($i$ and $j$) as a parameter $c_{ij}$. *Performance coupling* ($c_{ij}$) refers to the effect that kernel $i$ has on $j$ in relation to running each kernel in isolation.

We can group the parameters into three categories:

- $c_{ij} = 1$ indicates no interaction between the two kernels, yielding no change in performance.
- $c_{ij} < 1$ results from some resource(s) being shared between the kernels, producing a performance gain.
- $c_{ij} > 1$ occurs when the kernels interfere with each other, resulting in a performance loss.

Therefore, it should be the goal to use code that minimizes $c_{ij}$ to achieve best performance.

To compute the parameter $c_{ij}$, three measurements must be taken:

1. $p_i$ is the performance of kernel $i$ alone.
2. $p_j$ is the performance of kernel $j$ alone, and
3. $p_{ij}$ is the performance of kernels $i$ and $j$ together.

The above values can measure the usage of any resource that can improve the performance of an application through sharing or reuse. Resources that might be measured are caches, interconnection networks, and storage media. This work looks at the cache usage by measuring misses at the various levels. Also, because $c_{ij}$ represents the direct interaction between two kernels, the only measurements needed for $p_{ij}$ are those that correspond to consecutive kernels that occur in the application; there is no need to calculate $c_{ij}$ for all pairs of kernels. Without any interaction between kernels, we expect $p_{ij}$ to be the sum of $p_i$ and $p_j$. Since $c_{ij}$ is the measurement of interaction between the kernels, we compute it as the ratio of the actual performance of the kernels together to the no interaction expectation. (i.e., $c_{ij} = \frac{p_{ij}}{p_i+p_j}$).

Given the values of $p_i$, $p_j$, and $p_{ij}$, we represent the performance of the full application as a weighted, directed acyclic graph (DAG). To construct the graph, we create one node for each kernel in the application and a directed edge from node $i$ to node $j$ if kernel $i$ immediately precedes kernel $j$ in the application. Each node $i$ is given the weight $p_i$ and each edge from node $i$ to node $j$ is given the weight $c_{ij}$. By traversing the graph from the initial kernel to the final kernel, we can approximate the performance of the entire application. A naive approach would estimate total program performance by accumulating the weight of the nodes along the path. Consideration of the coupling as well as the kernel performance requires the following sum of products: $p_1 + \sum_{i=2}^{kernels} c_{i-1i} \times p_i$.

Evaluation of multiple algorithms for a given kernel entails including separate nodes for each algorithm and measuring the necessary interactions. Then, identifying the best algorithm for a given kernel requires determining the shortest path from the initial program segment to the final segment. An example of such a DAG is in Figure 1 and will be described further in Section 5.1.

### 3.2 Machine Description

For the experiments, we used the SGI Origin2000 at Northwestern University in the Center for Parallel and Distributed Computing. It is an 8 processor cache coherent non-uniform memory access machine with 1GB main memory; each processor is a 64 bit chip running at 195 MHz capable of 390 Mflops (2 flops per cycle)[9]. Each processor has 64 floating point registers and 64 integer registers. The machine has separate level one instruction and data caches, but a unified level two cache. The level one caches are 32KB each with two-way set associativity. The instruction cache has a line size of 64 bytes, while the data cache line size is smaller at 32 bytes. The 4MB level two cache is two-way set associative with a 128 byte line size.

The processors also have counters on chip that are able to count various events including cache misses. Counting these events costs very little because the hardware generates an interrupt after a certain value is reached in the counter (2053 for level one data cache). The software that runs to satisfy the interrupt increments a count value stored in memory along with the program location of the interrupt occurrence. This count value, when multiplied by 2053 accurately represents the number of cache misses.

## 4 Illustration of Concepts

To illustrate the ideas behind the coupling parameter and to verify the model, we use a synthetic program with two kernels that generate specified data streams. For these experiments, the kernels iterate through different sized arrays with a stride of one.

It is very easy to predict the number of misses for the synthetic program. We know the

exact access patterns and the cache characteristics of the machine, which can be combined to form a prediction: for arrays smaller than the cache, the number of misses is the number of misses needed to pull the array into the cache, or $\frac{array\_size}{cache\_line\_size}$, and for arrays larger than the cache, each cache line must be refetched because it was flushed by another array location, thus the total number of misses are $\frac{num\_accesses}{cache\_line\_size}$.

To illustrate the predictability of the synthetic program, we ran a number of experiments to measure the first level cache misses for the two different options of the kernels with array sizes of 4KB to 512KB. Each array element is four bytes. The results are given in Table 1 except for sizes less than 64KB that do not generate any data. The hardware counters do not generate an interrupt until 2053 cache misses occur, and the predicted number of misses for the array sizes smaller than 64KB is less than 2053.

TABLE 1
*Level one cache misses*

| Size | Predicted | Measured | Relative Error |
|---|---|---|---|
| 512KB | 6553600 | 6555229 | -0.00024850 |
| 256KB | 3276800 | 3276588 | 0.00006470 |
| 128KB | 1638400 | 1640347 | -0.00118694 |
| 64KB | 819200 | 817094 | 0.00257742 |

The predicted column is computed as described previously. The last column is the fraction of relative error. By observing the final column, we can see that the synthetic program performs as predicted.

After measuring each kernel in isolation, experiments were conducted to measure the synthetic program with different array sizes for each of the two kernels. The results, then, led to the generation of the coupling parameter for each run. These experiments illustrate what the coupling parameter represents through a series of three experiments. The first set of experiments considers the kernels accessing arrays with the same starting address, but different sizes. The last two experiments demonstrate how the coupling parameter changes when the starting address of the arrays are not the same.

Table 2 has the results for the first experiment for which the arrays have the same starting address. Column one identifies the array sizes used for each kernel. The second column gives the cache misses for the first kernel, and the third column for the second kernel. The two columns can be compared to the measured values in Table 1 to identify if the cache misses increased (destructive coupling) or decreased (constructive coupling) when the two kernels were executed together as compared to when the two kernels were executed individually.

TABLE 2
*overlapping arrays*

| Kernels | Kernel 1 misses | Kernel 2 misses | Coupling Parameter |
|---|---|---|---|
| 4KB $\Rightarrow$ 128KB | 57484 | 1582863 | 1.0 |
| 8KB $\Rightarrow$ 128KB | 626165 | 1014182 | 1.0 |
| 16KB $\Rightarrow$ 128KB | 427024 | 1211270 | 0.99874843 |
| 32KB $\Rightarrow$ 128KB | 254572 | 1412464 | 1.00495049 |

In Table 2, the misses for kernel two (128KB) decreased in all cases as compared to Table 1; however, the misses for kernel one increased in all cases. When run in isolation, the first kernel is able to load the entire array into the cache and only incur misses for the first time each element is read. When run with the second kernel, however, the first kernel must reload the data into the cache because the second kernel flushes the cache every iteration, causing destructive coupling. Conversely, when run in isolation, the second kernel always had to reload the array into the cache. When run with the first kernel, however, the second kernel can access the array that the first kernel has already loaded into the cache causing a smaller number of misses, or constructive coupling. In this example, the coupling parameter remains at 1.0 because the constructive coupling is exactly the same as the destructive coupling, thereby offsetting each other. Hence, the coupling parameter measures effective performance.

The second experiment explores the impact of moving the starting address of the array in the first kernel such that the data left in the cache by the second kernel is immediately accessed by the first kernel on the next iteration. We expect this alignment to produce constructive coupling by reducing the number of misses by the first kernel.

TABLE 3
*partially overlapping arrays*

| Kernels | Kernel 1 | Kernel 2 | Coupling Parameter |
|---|---|---|---|
| 64KB ⇒ 128KB | 457819 | 1640347 | 0.85380116 |
| 128KB ⇒ 128KB | 1276966 | 1640347 | 0.88923654 |
| 256KB ⇒ 128KB | 2917313 | 1642400 | 0.92734864 |
| 512KB ⇒ 128KB | 6193901 | 1640347 | 0.95591182 |

In Table 3, the misses for kernel one decreased as compared to Table 1. When run in isolation, kernel one was forced to reload its array into the cache each iteration because of the larger array sizes. By reusing the data left by kernel two in the cache, kernel one does not incur as many cache misses causing constructive coupling. Since there is no destructive coupling to balance the constructive coupling, the coupling parameter is less than 1.0. As the size of kernel one increases, however, the coupling parameter increases, since the amount of reuse become a smaller percentage of the total number of accesses. This result is significant. We have not altered either kernel to change the number of cache misses when run in isolation. The only reason for the decrease in cache misses in kernel one is due to kernel two loading the data that kernel one needs. This coupling can only be measured when they are run together–never separately.

The last experiment again looks at the impact of moving the starting address of the array in the first kernel such that the data accessed by the two kernels do not overlap. This "mis-alignment" can only cause destructive coupling by causing extra misses, since the two kernels will never share resources.

In Table 4, the misses for kernel one increased as compared to Table 1. Both kernels are forced to reload all of their data into the cache during each iteration causing destructive coupling. This results in the coupling parameter being greater than 1.0. The coupling increases as the size of kernel one increases since kernel one must reload more data as its size increases.

The three experiments illustrate the concepts behind the value of the coupling parameter using a synthetic benchmark. In particular, the experiments illustrate how

TABLE 4
*non-overlapping arrays*

| Kernels | Kernel 1 | Kernel 2 | Coupling Parameter |
|---|---|---|---|
| 4KB ⇒ 128KB | 53378 | 1636241 | 1.03003754 |
| 8KB ⇒ 128KB | 102650 | 1640347 | 1.06257822 |
| 16KB ⇒ 128KB | 203247 | 1642400 | 1.12515644 |
| 32KB ⇒ 128KB | 410600 | 1640347 | 1.23638613 |

the coupling parameter can change to be destructive or constructive without changing the the isolated performance of the kernels. The use of the coupling parameter with a widely used application is described below.

## 5 Experimental Results

### 5.1 Application Description

The conjugate gradient (CG) benchmark solves the equation $Ax = b$ using an iterative process to search through the solution space. The main computational complexity occurs during the matrix-vector multiply section of the code. We divide the CG application into three kernels:

1. **Initialization** sets up the data structures after randomly generating the nonzero elements in the sparse matrix.

2. **Matrix-Vector Multiply** performs the largest piece of computation that can have varying numbers of cache misses based on the data structure that stores the $A$ matrix.

3. **Remaining Vector Operations** performs the rest of the computations needed to perform the conjugate gradient application and then any cleanup after the application is finished.

The execution of the program performs the initialization kernel once and then iterates through the other two. This results in possible couplings between $1 \Rightarrow 2$, and $2 \Leftrightarrow 3$.

### 5.2 Level 1 Cache misses

We considered three different sparse matrix representations (CMNS [4], SPAR [12], and ITPACK [5]) in addition to the original representation [2] for the matrix-vector multiply kernel. The results of measuring the isolated performance of each kernel and the coupling parameters are given in Figure 1. Initial inspection of Figure 1 immediately rules out the ITPACK representation because of its poor performance (42784520 misses vs. 12933900 misses), which is not compensated by a small enough coupling parameter. This leaves CMNS, SPAR, and the original representation as possibilities. CMNS and SPAR have destructive interaction with the remainder of the code ($c_{ij} > 1.0$), whereas the original code has approximately no coupling with the remainder of the code ($c_{ij} = 1.02$). If one considers cache misses only, which is the traditional method, one would select the original representation or CMNS. Considering the coupling parameter, however, results in the CMNS representation also being eliminated, and the original representation being selected. In terms of level one cache misses, original has 22028 fewer misses than CMNS.

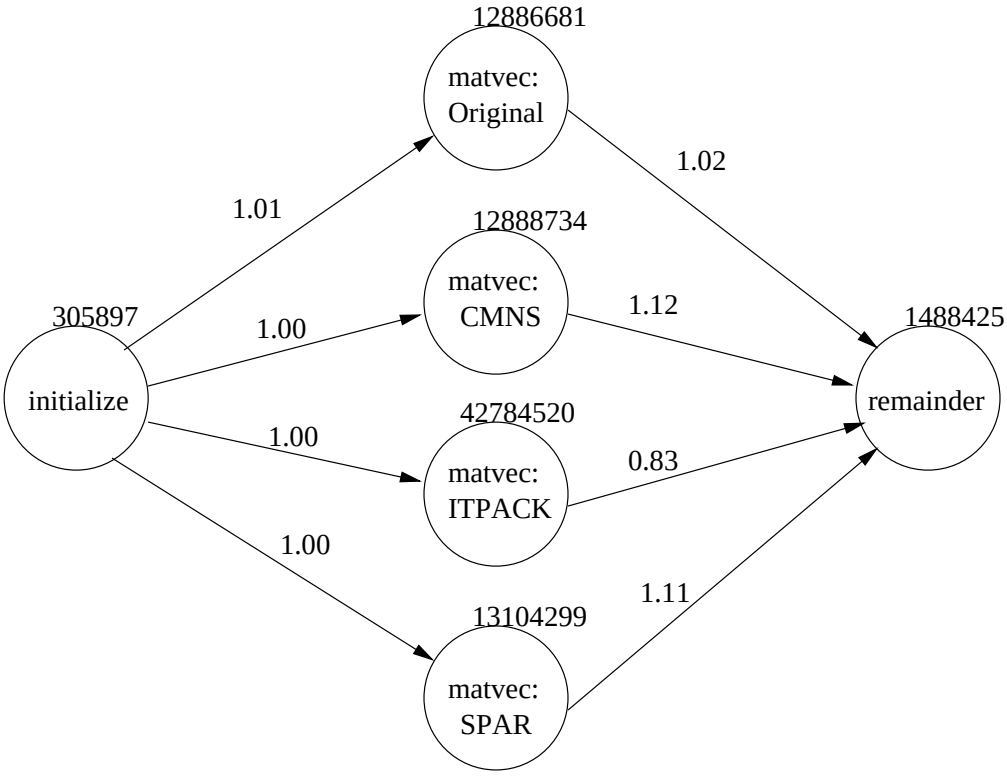


FIG. 1. *Conjugate gradient weighted directed acyclic graph*

### 5.3 New Algorithm Development

Next, we studied the code carefully to understand why the various coupling parameters existed[1]. In particular, we studied the code to identify why ITPACK had such a low coupling parameter. The code for ITPACK is in Table 6 in the appendix. As can be seen by the code, the $\vec{w}$ vector is strided through during each iteration of the outside loop. In the CG code, however, the first vector used is the $\vec{w}$ vector. ITPACK and CMNS force the $\vec{w}$ vector to be left in the cache causing fewer cache misses. The other data structures only iterate through the $\vec{w}$ vector once (original) or in a non-strided manner (SPAR). The problem with ITPACK, however, is that it accesses too many zero entries in the compressed sparse matrix, causing a significant number of misses.

Further examination of the conjugate gradient code reveals that the last vector used before the matrix-vector multiply is the $\vec{p}$ vector. Both ITPACK and the original code access $\vec{p}$ in a non-strided manner that benefits from having the $\vec{p}$ vector in the cache. In contrast, CMNS and SPAR only iterate through the $\vec{p}$ vector once, only the first few accesses of $\vec{p}$ will hit the cache before $\vec{p}$ is flushed by other accesses.

The aforementioned facts suggest the following characteristics for a hybrid code for matrix-vector multiply:

1. The code should iterate through the $\vec{w}$ vector to keep $\vec{w}$ in cache for use by the remainder of the code when the matrix-vector multiply finishes (similar to ITPACK and CMNS).

[1]For a full description of each data structure and the algorithm to compute with, see the appendix.

2. The code should iterate through the $\vec{p}$ vector frequently to take advantage of $\vec{p}$ being left in the cache prior to the start of the matrix-vector multiply (similar to original and ITPACK).

3. The data structure should not contain zero entries.

The above characteristics suggest a code for matrix-vector multiply such that the first part uses $\vec{p}$ frequently (to take advantage of $\vec{p}$ being left in the cache by the remainder of the code) and the latter part strides through $\vec{w}$ to leave $\vec{w}$ in the cache to be used by the remainder of the code. This was achieved by splitting a matrix in half. The first N/2 columns of the matrix were stored row-wise, using the original data structure. The second N/2 columns of the matrix were stored column-wise using CMNS. Hence the hybrid data structure does not store any zero entries.

The new algorithm resulted in 12898999 misses for the matrix-vector multiply and a coupling parameter of 1.0. The number of misses in in the range of original and CMNS, but the coupling is better. The reduction in coupling for the new algorithm is enough such that the new algorithm has 17451 fewer cache misses (0.995% less) than the original data structure. While the reduction is only 1%, the example demonstrates how the methodology is used in the development of a new data structure that had good performance.

## 6 Analysis and Summary

In this paper we presented a methodology for quantifying and understanding the interaction between kernels. The interaction was represented as the coupling parameter, $c_{ij}$, which measured the effective interaction between adjacent kernels in an application. We illustrated the concepts behind the coupling parameter using a synthetic benchmark. This benchmark demonstrated how coupling can be changed without affecting the isolated performance.

Further, we used the coupling parameter with the CG application. The CG example illustrates two uses for the coupling parameter:

- The coupling parameter can be used to make performance directed By considering the performance of a kernel as well as its coupling with other kernels, we are able to determine which algorithm will have the best performance for a given kernel. The coupling parameter illuminates how potential algorithms will interact with the overall application.

- The coupling parameter can be used to help understand how kernels interact, for which this understanding can lead to better hybrid algorithms. The phenomenon was illustrated with the CG algorithm to provide slight improvement over existing algorithms.

## 7 Future Work

Future work includes analyzing more applications and different platforms. Further, we plan to extend this work to parallel architectures.

## References

[1] Bowen Alpern, Larry Carter, and Jeanne Ferrante. Modeling parallel computers as memory hierarchies. In *Conference on Programming Models for Massively Parallel Computers*, 1993.

[2] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA, December 1995.

[3] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Practices and Principles of Parallel Programming*, pages 1–12, May 1993.

[4] A. George and J. Kiu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[5] D.R. Kincaid, J.R. Respess, D.M. Young, and R.G. Grimes. Itpack 2c: A fortran package for solving large sparse linear systems by adaptive accelerated iterative methods. *ACM Transactions on Mathematical Software*, 8:302–322, 1982.

[6] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[7] Rafael H. Saavedra and Alan Jay Smith. Analysis of benchmark characteristics and benchmark performance prediction. Technical Report CSD-92-715, University of California, Berkeley, 1992.

[8] Rafael H. Saavedra and Alan Jay Smith. Measuring cache and TLB performance and their effect on benchmark run times. Technical Report CSD-93-767, University of California, Berkeley, 1993.

[9] Subhash Saini and David Bailey. Hot chips for high performance computing. In *SuperComputing Tutorials*, November 1996.

[10] Anand Sivasubramaniam, Umakishore Ramachandran, and H. Venkateswaran. Message-passing: Computational model, programming paradigm, and experimental studies. Technical Report GIT-CC-91/11, Georgia Institute of Technology, February 1991.

[11] Anand Sivasubramaniam, Umakishore Ramachandran, and H. Venkateswaran. A comparative evaluation of techniques for studying parallel system performance. Technical Report GIT-CC-94/38, Georgia Institute of Technology, September 1994.

[12] Valerie E. Taylor, Abhiram Ranade, and David G. Messerschmitt. SPAR: A new architecture for large finite element computations. *IEEE Transactions on Computers*, 44(4):531–545, April 1995.

[13] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

## A Conjugate Gradient Algorithms and Representations

The following demonstrates the algorithms used in the conjugate gradient section along with an example representation of the matrix:

$$K = \begin{bmatrix} k_{11} & 0 & k_{13} & 0 & 0 \\ 0 & k_{22} & 0 & 0 & k_{25} \\ k_{31} & 0 & k_{33} & 0 & k_{35} \\ 0 & 0 & 0 & k_{44} & 0 \\ 0 & k_{52} & k_{53} & 0 & k_{55} \end{bmatrix}$$

### A.1 CMNS

The Column-Major Nonzero Storage (CMNS) algorithm uses three one-dimensional vectors. The nonzeros are stored in the vector $\vec{K}_v^T$. $\vec{L}^T$ stores the size of each row, and $\vec{R}^T$ indicates the row for each corresponding element in $\vec{K}_v^T$. For example, matrix $K$ is stored as:

$$\vec{K}_v^T = \left[ \underbrace{k_{11}, k_{31}}_{column1}, \underbrace{k_{22}, k_{52}}_{column2}, \underbrace{k_{13}, k_{33}, k_{53}}_{column3}, \underbrace{k_{44}}_{column4}, \underbrace{k_{25}, k_{35}, k_{55}}_{column5} \right]$$
$$\vec{R}^T = [1, 3, 2, 5, 1, 3, 5, 4, 2, 3, 5]$$
$$\vec{L}^T = [2, 2, 3, 1, 3]$$

TABLE 5
*CMNS algorithm*

```
index = 1
for column = 1 to N
    repeat
        w[R[index]] += Kv[index] * p[column]
        index += 1
    L[column] times
end for
```

## A.2 ITPACK

The ITPACK algorithm uses two two-dimensional matrices. Each matrix has a width of the row with the most nonzeros in the original matrix and height identical to the original matrix. Each row of the original matrix is compacted in $K_v$ with extra entries set to 0.0. $C$ stores the column for each corresponding value. For example, matrix $K$ is stored as:

$$K_v = \begin{bmatrix} k_{11} & k_{13} & 0.0 \\ k_{22} & k_{25} & 0.0 \\ k_{31} & k_{33} & k_{35} \\ k_{44} & 0.0 & 0.0 \\ k_{52} & k_{53} & k_{55} \end{bmatrix} \quad C = \begin{bmatrix} 1 & 3 & x \\ 2 & 5 & x \\ 1 & 3 & 5 \\ 4 & x & x \\ 2 & 3 & 5 \end{bmatrix}$$

TABLE 6
*ITPACK algorithm*

```
for column = 1 to M
    for index = 1 to N
        w[index] += Kv[index][column] * p[C[index][column]]
    end for
end for
```

## A.3 SPAR

The SPAR algorithm uses two one-dimensional vectors. As in CMNS, $\vec{K}_v^T$ contains the columns of the original matrix, but the columns are separated by zeros. In the $\vec{R}^T$, the values that correspond to the zeros are the column of the next set of data. For example,

matrix $K$ is stored as:

$$\vec{K}_v^T = \left[\underbrace{k_{11}, k_{31}}_{column1}, \mathbf{0.0}, \underbrace{k_{22}, k_{52}}_{column2}, \mathbf{0.0}, \underbrace{k_{13}, k_{33}, k_{53}}_{column3}, \mathbf{0.0}, \underbrace{k_{44}}_{column4}, \mathbf{0.0}, \underbrace{k_{25}, k_{35}, k_{55}}_{column5}\right]$$
$$\vec{R}^T = [1, 3, \mathbf{2}, 2, 5, \mathbf{3}, 1, 3, 5, \mathbf{4}, 4, \mathbf{5}, 2, 3, 5]$$

TABLE 7
*SPAR algorithm*

```
column = 1
for index = 1 to nz + N - 1
    if (Kv[index] = 0.0) then
        column = R[index]
    else
        w[R[index]] += Kv[index] * p[column]
    end if
end for
```

## A.4 Original

The original algorithm uses three one-dimensional vectors. The $\vec{A}^T$ vector contains the nonzeros stored rowwise. The $\vec{L}^T$ vector contains indices into the startpoints of each row, so $\vec{L}[i]$ indicates which element in $A$ starts row $i$. Finally, $\vec{C}^T$ indicates the column for the corresponding entry in $A$. For example, matrix $K$ is stored as:

$$\vec{A}^T = \left[\underbrace{k_{11}, k_{13}}_{row1}, \underbrace{k_{22}, k_{25}}_{row2}, \underbrace{k_{31}, k_{33}, k_{35}}_{row3}, \underbrace{k_{44}}_{row4}, \underbrace{k_{52}, k_{53}, k_{55}}_{row5}\right]$$
$$\vec{L}^T = [1, 3, 5, 8, 9]$$
$$\vec{C}^T = [1, 3, 2, 5, 1, 3, 5, 4, 2, 3, 5]$$

TABLE 8
*Original algorithm*

```
for row = 1 to N
    for index = L[row] to L[row + 1] - 1
        w[row] += A[index] * p[C[index]]
    end for
end for
```

### A.5 Hybrid

We have split $K$ so that the first two columns are stored row-wise and the last three are stored column-wise. Matrix $K$ is stored as:

$$\vec{A}^T = \left[ \underbrace{\overbrace{k_{11}}^{row1}, \overbrace{k_{22}}^{row2}, \overbrace{k_{31}}^{row3}, \overbrace{k_{52}}^{row5}}_{rowwise}, \underbrace{\overbrace{k_{13}, k_{33}, k_{53}}^{column3}, \overbrace{k_{44}}^{column4}, \overbrace{k_{25}, k_{35}, k_{55}}^{column5}}_{columnwise} \right]$$

$$\vec{rL}^T = [1, 1, 1, 0, 1]$$
$$\vec{C}^T = [1, 2, 1, 2]$$
$$\vec{cL}^T = [3, 1, 3]$$
$$\vec{R}^T = [1, 3, 5, 4, 2, 3, 5]$$

TABLE 9
*New Hybird algorithm*

```
index = 1
for row = 1 to N
    repeat
        w[row] += A[index] * p[C[index]]
        index += 1
    rL[row] times
end for

for column = N/2 to N
    repeat
        w[R[index]] += A[index] * p[column]
        index += 1
    cL[column] times
end for
```